Chapter 1
Novice Programming Environments:
Lowering the Barriers, Supporting the Progression

Judith Good
University of Sussex, UK

ABSTRACT

In 2011, the author published an article that looked at the state of the art in novice programming environments. At the time, there had been an increase in the number of programming environments that were freely available for use by novice programmers, particularly children and young people. What was interesting was that they offered a relatively sophisticated set of development and support features within motivating and engaging environments, where programming could be seen as a means to a creative end, rather than an end in itself. Furthermore, these environments incorporated support for the social and collaborative aspects of learning. The article considered five environments—Scratch, Alice, Looking Glass, Greenfoot, and Flip—examining their characteristics and investigating the opportunities they might offer to educators and learners alike. It also considered the broader implications of such environments for both teaching and research. In this chapter, the author revisits the same five environments, looking at how they have changed in the intervening years. She considers their evolution in relation to changes in the field more broadly (e.g., an increased focus on "programming for all") and reflects on the implications for teaching, as well as research and further development.

DOI: 10.4018/978-1-5225-5969-6.ch001

INTRODUCTION

In 2011, an article I wrote, entitled, "Learners at the wheel: Novice programming environments come of age" was published in the International Journal of People Oriented Programming (IJPOP). It is interesting to see how things have evolved in the intervening six years. In some cases, there have been substantial advances in terms of novice programming environments, as well as the computational thinking agenda and computer science education in general, whilst in others, the issues identified as relevant then are equally relevant now.

In the original article, I stated:

Over the past few years, a number of programming environments for novices have moved out of the research lab and into the public domain. Many of these environments are available free for download, and learners can begin using them to create simple programs in a matter of minutes. This is an exciting trend for a number of reasons: firstly, the environments have increased significantly in terms of their sophistication, combining programming languages (either graphical or text-based) with 2D and sometimes 3D graphical execution environments to form fully fledged integrated development environments (IDEs). Secondly, the IDEs themselves are often embedded in what could be considered a broader ecosystem, comprising online peer support facilities, educational resources for both teachers and learners, and mechanisms for sharing the programs that one has created with other learners. And finally, many of these environments, whilst being open-ended in scope, and allowing for user creativity, are nonetheless grounded in motivating activities such as game making, animation, storytelling, etc. This is not to suggest that novice programming environments are a new phenomenon; indeed, Guzdial (2004) provides an overview of their history since the 1960s, while Kelleher and Pausch (2005) have developed one of the most extensive taxonomies to date. However, because of the ease with which the World Wide Web can make such programming environments and their associated infrastructures so freely accessible, environments of this type are much more ubiquitous, with sites such as Scratch (http://scratch.mit.edu/) reporting over half a million registered users, and over one million uploaded projects.

In particular, children and young people are finding themselves willingly engaged in programming in the pursuit of other creative activities such as making games, interactive stories, simulations or animated films. While they would not classify themselves as programmers, nor would many consider pursuing a career in computer science, they nonetheless enjoy the creative process of designing an artefact and bringing it to life, as it were, by giving it interactivity and, at a later stage, sharing it with one's peers, both locally and remotely, often with great enthusiasm. As such,

these environments have opened up new worlds to novice programmers and, more informally, to "unwitting end user programmers" (Petre & Blackwell, 2007). Whereas end user programmers do not program on a regular basis, but might occasionally write a small program to achieve a particular goal, unwitting end user programmers may be unaware that what they are doing is even programming at all, a phenomenon also observed by Resnick et al. (2009) in the case of young people using Scratch.

What is perhaps most exciting about these novice programming environments is that, examined in a broader context, they may offer some answers to questions currently being posed by educators, namely, with computation becoming ever more ubiquitous and pervasive, how do we ensure that future generations are conversant with computational tools, not just as consumers, but as producers? Although learning to think 'computationally' has long been recognised as important (Papert, 1980), the recent computational thinking drive has refocused attention on this as a significant issue in modern society (Wing, 2006). There is broad agreement that it is important to teach computational thinking skills from a young age, and to people who may never learn to program (Guzdial, 2008; Fletcher & Lu, 2009), but deciding which specific skills should be taught is still an emerging endeavour.

In this article, I will argue that current novice programming environments, and in particular, the ways in which they have already been appropriated by young people, may provide some ideas for educators interested in looking at how to teach computational skills to people outside of traditional computer science disciplines. I describe the current state of the art in programming environments for novices by considering five such environments. The aim is not to provide an exhaustive review, but to highlight various features which promote learning, ease of use and engagement. I go on to discuss the features they share, and highlight some of the implications of these environments for both teaching, and research.

In this chapter, I will revisit the five novice programming environments described in my original article: Scratch, Alice, Looking Glass, Greenfoot and Flip. All have evolved substantially in the intervening years, and all are still in current use (with the exception of Flip which, while it would be technically possible to use, has suffered from being tied to a commercial games engine which is now outdated, as well as from a lack of follow up funding). Indeed, in terms of those environments which continue to receive funding, their growth and reach has increased dramatically: to date, over 20 million Scratch projects have been shared by young people through the Scratch online community (Resnick, 2017). The infrastructure around these environments has also expanded significantly, with the environment now just one element of a comprehensive set of resources for both educators and learners, as

well as opportunities to become part of virtual communities, join discussions, share creations, etc.

In addition to revisiting the five environments above, I will also, briefly, consider new novice programming environments which have become available in the intervening years.

NOVICE PROGRAMMING ENVIRONMENTS

In this section, I consider five very different programming environments for novices: Scratch, Alice, Looking Glass, Greenfoot and Flip. All vary quite significantly on a number of aspects, and as such, have differing strengths and weaknesses. Only a cursory overview can be provided for each environment, however as most of the environments are quite well established, full references are given for each (including links to websites) so that they can be followed up if the reader so wishes. I consider basic information about the environment, such as the age range for which it was designed, the programming language on which it is built (if relevant), and whether it has evolved from previous versions or other languages. I then look at the *process* of creating a program in the environment, describing a typical program creation session. I also discuss, for each environment, any special features which mark it out from the others, and extra support and community features which enhance the standard programming environment. Finally, I consider the ways in which each environment is currently being used in the wild.

Scratch

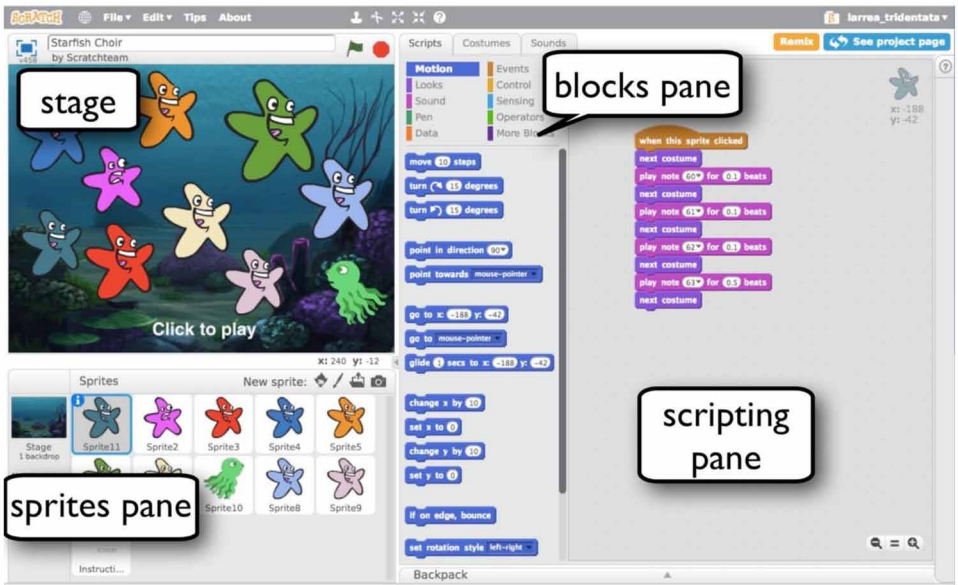
Scratch (http://scratch.mit.edu/) is a programming environment designed to allow novices to manipulate media through programming activities (Maloney et al., 2008), with a primary audience aged between 8 and 16 (Resnick et al., 2009). Scratch 2, the current version of Scratch, was released in 2013, with Scratch 3 expected in 2018. Scratch has evolved from, and been inspired by, a number of other languages and environments developed in the MIT Media Lab, and is designed in such a way as to promote playful tinkering and exploration. Compared to the other environments surveyed, Scratch allows a younger age range to begin experimenting with programming concepts. An additional programming environment, ScratchJr (released in 2014), has been designed to further lower the age barrier for computing, and is aimed at children aged 5-7. Although ScratchJr shares some features with Scratch, it crucially does not require reading skills in order to write programs. A fuller review of the ScratchJr environment can be found in (Goschnick, 2015).

The Scratch interface (Figure 1) comprises a blocks pane, in the middle of the interface, with ten different categories of colour coded programming blocks shown at the top of the pane (with Scratch 2 having introduced the ability to create custom blocks). The blocks themselves snap together to create a program, with shape being indicative of function, e.g. a "forever" block will fit round a set of commands and cause those commands to run indefinitely. Programs are created by dragging blocks from the blocks pane in the middle to the scripting pane on the right-hand side of the screen. The scripting pane can accommodate multiple sets of blocks, thus creating de facto concurrent programs with ease.

On the top left-hand side of the interface is the stage, which can be maximised once a project is finished. Sprites (selected from the sprites pane) appear on the stage, and their behaviours are governed by the programs created in the scripting pane. In contrast to previous versions of the interface, the Scratch 2 interface has been streamlined to allow more space for creating programs, with blocks, costumes and sounds accessed via tabs in the middle part of the interface.

The concept of an online community of users has been central to Scratch from its inception, with the idea of sharing programs given the same importance as creating them in the first place. The Scratch website acts as a vibrant online space in which learners can share projects they have created with others, who can rate those projects, download them, remix them, and re-upload them if they wish. In a very novel approach to fostering the collaborative aspects of code reuse, and ensuring that

Figure 1. The Scratch interface



the original authors retain credit, the Scratch website includes a "remix tree", which allows users to see who has remixed their project and in what ways and, similarly, how their remixed project was subsequently remixed by others (Resnick, 2014).

Teachers wishing to use Scratch also have access to their own online community, ScratchEd, where they can ask questions and get advice. Additionally, the Scratch website provides numerous lesson plans, projects and educational resources which can be freely downloaded.

Interestingly enough, studies of young people using Scratch show that, despite the absence of formal instruction or even mentors experienced in computer science concepts, they still produce projects which contain many of the hallmarks of traditional programs, e.g. conditionals, variables and Boolean logic (Maloney et al., 2008). Despite this, when interviewed, they do not consider themselves to be "programming", although the term 'coding' is becoming used more frequently to denote these types of informal programming activities. Recent research has also shown the viability of Scratch as an effective platform for teaching important computer science concepts, even if students experience difficulties with concepts such as variables, concurrency and repeated execution (Meerbaum-Salant et al., 2013). Although originally designed for young people of school age, research using Scratch at university level suggests that it increases students' self-efficacy, and leads to much greater success when students go on to learn object-oriented programming in CS1 (Rizvi et al., 2011).

Alice

Alice (http://www.alice.org/) is a development environment designed to teach programming through the building of 3D virtual worlds (Cooper, Dann & Pausch, 2003; Adams, 2014). Alice can be used by students in middle school (typically 11 years old and upwards) through to university as it supports the development of reasonably substantial programs, e.g. 3000 lines of code (Kelleher et al., 2002). Like the environments mentioned above, Alice has been in continuous development since it was released, with Alice 3.3 being the current version at the time of writing. Note: The founder of the Alice Project at Carnegie Mellon was Dr. Randy Pausch, who presented The Last Lecture in 2007, a year before his death.

Alice is designed around an object-oriented paradigm, and uses a drag and drop interface where users can drag program tiles (now called Controls) into a code editor and parameterise them if necessary. Once a program has been constructed, it is executed in a 3D world, allowing the learner to very quickly see whether or not her program implemented the desired behaviour. In the top left panel is the Camera view with two overlaid buttons: the Run button executes the programmed animation as it currently stands; while the Scene button takes the user into Scene

Editor mode, depicted in Figure 3. The opening interface is dominated by the Edit panel (right) itself split in two with Alice code in the left half, and the equivalent Java code displayed in the right half. This ability to display the Java code equivalent is a non-default option, included by the Alice developers to help coders transition from the drag and drop Alice code, to a mainstream commercial programming language. The Edit panel has tabbed panes where different parts of a program are created and edited. Code Editor mode also has a Methods panel (bottom left) and a Controls panel (along the bottom right). The Methods panel has two tabs, one for Procedures and the other for Functions. Alice differentiates the two as follows: *Procedures* perform actions on or by an object; while *Functions* ask a question of the user or compute a value and so on. In Java, both are just called methods.

When Alice is first started with a new project template, the Camera is the current object, representing the Camera view currently displayed in top right of Figure 2. Different objects within the scene can be selected and worked on, for example, an object called Penguin is currently being edited. All objects in Alice have a method called *myFirstMethod* (the main method defined for a scene), which is the default open tab in the Edit panel as depicted in Figure 2.

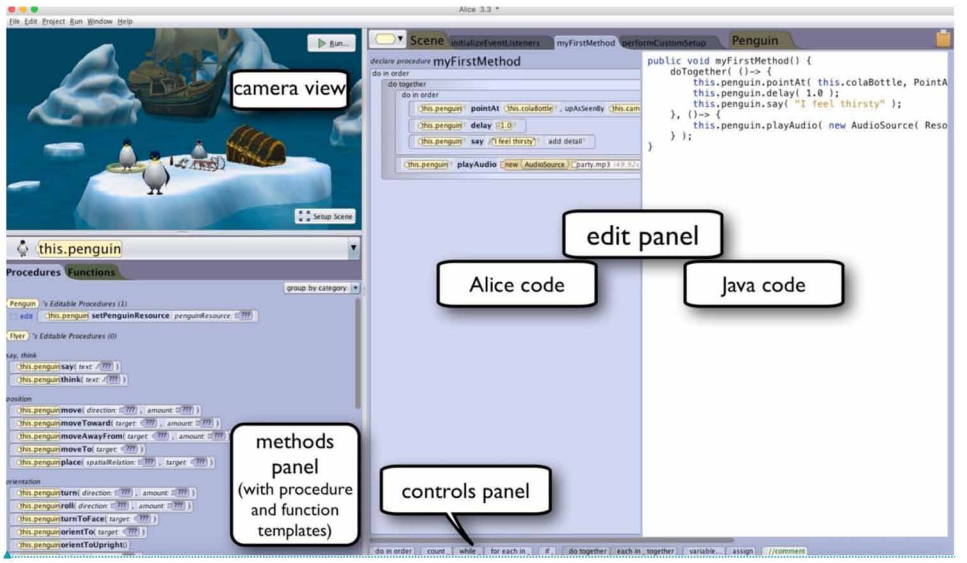
Code is developed by dragging and dropping the pre-built Procedure and Function templates from the Methods panel into the Edit panel, and then populating those methods further by dragging and dropping *Controls* from the Control panel. Each control has some parameters within it that can be edited by the coder. In this sense, the Alice controls are very similar to the blocks in Scratch.

Alice was an early example of a programming environment that provided a motivational "hook", namely the ability to create one's own 3D animations. In addition, it gave learners a very tight visual feedback loop: it was very clear from the way the characters in the environment behaved (or not), whether the program created had had the desired effect, and its animated output was more compelling than a standard debugger.

Alice has evolved substantially over the years: as shown in Figure 2, it can now show the Java code side by side with the original Alice code, however, the Java code in the right-hand panel cannot be edited. In addition, the Controls themselves can be turned into Java equivalents (of the Alice controls), so that these 'Java controls' can be dragged and dropped into the methods. Furthermore, Alice 3 can be used with NetBeans, via a plugin, to convert Alice files in Java, and allow users to continue working on their programs in Java. However, once the Java code has been modified in NetBeans, the changes cannot be brought back into Alice – it is a one-way migration.

From a visual perspective, Alice 3 incorporates extensive 3D content and assets from the hugely popular Sims™ II video game, making learners' creations more similar in look and feel to the games that many of them regularly play. This

Figure 2. The Alice 3 interface



extensive gallery of objects is selectable via the five tabs in the bottom panel of the Scene Editor (see Figure 3). The included gallery is necessarily large, as each object comes pre-programmed with a set of Method and Function templates, such as the *moveForward* template for Penguin, which the coder can select then modify and expand as needed. The coder can also add new procedures of their own to an object (although this gets considerably more complex). The objects from the Sims are sophisticated in that they often have skeletal systems (e.g. the humans and animals), and the movement of the individual body-parts can be programmed although the coding to do so is, again, quite complex.

The Scene Editor makes it very easy to create 3D scenes and populate them with numerous objects from the Gallery. The placement, rotation and resizing of each object is very intuitive, making good use of both mouse and keyboard, while displaying visual handles to facilitate making the desired changes. The Scene Editor and the Code Editor are very complementary tools, for example: the hierarchy of the classes in the current project is accessible from the Code Editor (via the drop-down button that precedes the tabs in the Edit panel, and shown in Figure 4); while the hierarchy of instantiated objects in the Scene is accessible in the Scene Editor.

And similarly to the other environments described in this chapter, the Alice IDE is just one component, albeit the core one, of a substantial set of resources, including how-tos, exercises and projects, textbooks, a teacher listserv and an open forum. In the time that has passed since my earlier paper, both the terminology used within

Figure 3. The Scene Editor in Alice

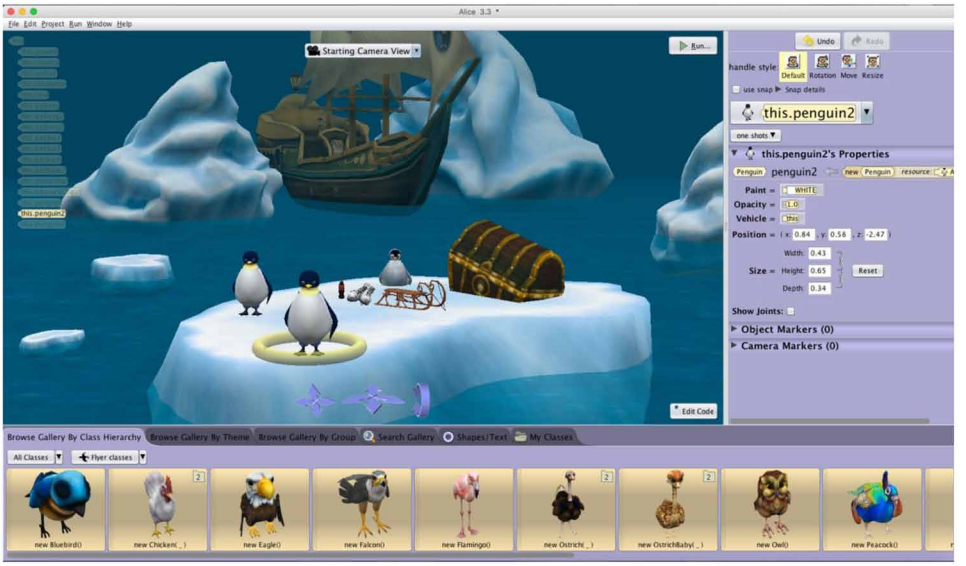
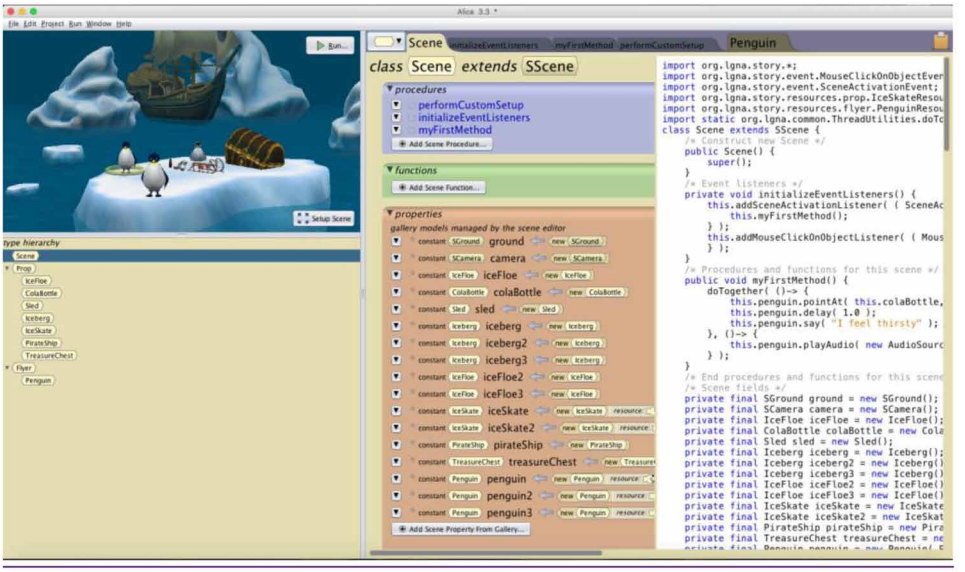


Figure 4. The Code Editor in Alice in Class mode, note: Methods Panel becomes class hierarchy



the interface and the inclusion of the view of the Java code behind the scenes, has gravitated Alice towards mainstream programming a little, by facilitating a transition for those first-time coders in Alice 3 who then wish to pursue coding in a mainstream object-oriented language within a mainstream IDE, namely Java in NetBeans.

Looking Glass

The Looking Glass IDE is the successor to Storytelling Alice, which was itself based on Alice 2.0. Whereas Alice relies on the creation of animations as its motivational hook for learning programming, Looking Glass is designed specifically with the aim of helping middle school girls (aged approximately 11-14 years) to create (non-interactive) stories in a 3D world, with the rationale that they will be sufficiently motivated by the activity to tackle the programming aspects as a result. This aim is set in the broader context of the large gender disparity in undergraduate computer science, and the need to address this gap as computing becomes ever more pervasive in society. Although the typical response has been to increase the use of video games in the undergraduate curriculum, Kelleher and Pausch (2007) have argued that this does not typically address female interests.

In order to provide support for storytelling activities, Alice was adapted in a number of ways. The resulting system contains animations which are specifically social and interactional (e.g. "look at", "touch" rather than "orient to"), 3D characters and objects designed to promote ideas for stories, and tutorials which use story-based examples rather than generic programming ones (Kelleher, Pausch & Kiesler, 2007).

The Looking Glass development environment consists of an Action Editor and a Scene Editor. In the Action Editor, learners can choose actions from the action list for each character (including the overall scene), drag the actions into the story pane, and set parameters for the actions, effectively creating the program (shown in Figure 5).

The Looking Glass IDE is built upon Alice, and has a Scene Editor very similar to that of Alice (see Figure 3). As noted above, in the Scene Editor, learners can choose 3D characters and objects from the gallery, and arrange them in the scene view in order to create their stories.

Like Scratch and Alice, Looking Glass promotes remixing, and provides explicit support for doing so. Learners can load a world created by another user, and then use video editing style controls to step through the code and select the parts they wish to retain (see Figure 6).

Again, like the other environments, Looking Glass has extensive accompanying resources and community features. In addition, the Looking Glass team have expended significant research effort into looking, in depth, at the best ways of scaffolding learning. For example, the team has looked at how tutorials can be automatically

Figure 5. The Looking Glass Action Editor

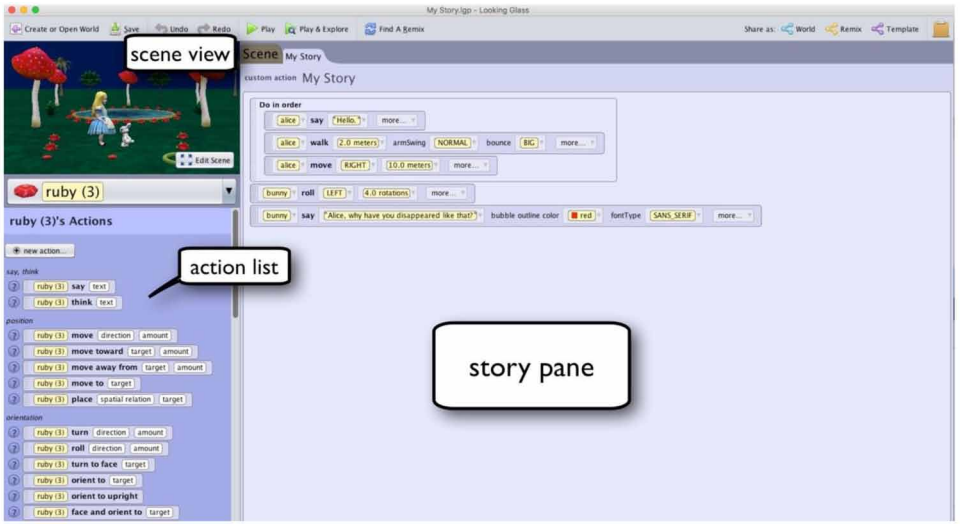
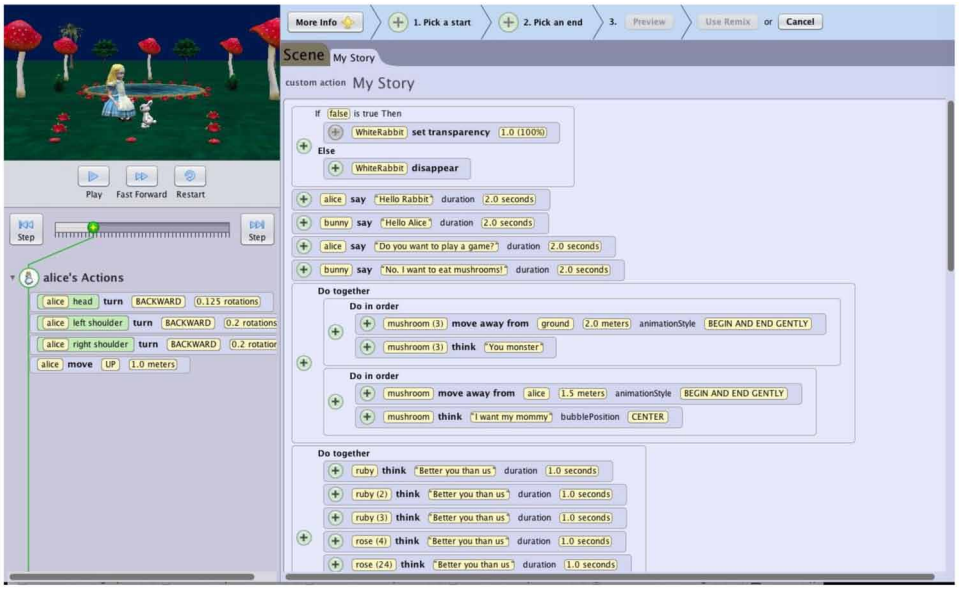


Figure 6. Support for remixing in Looking Glass

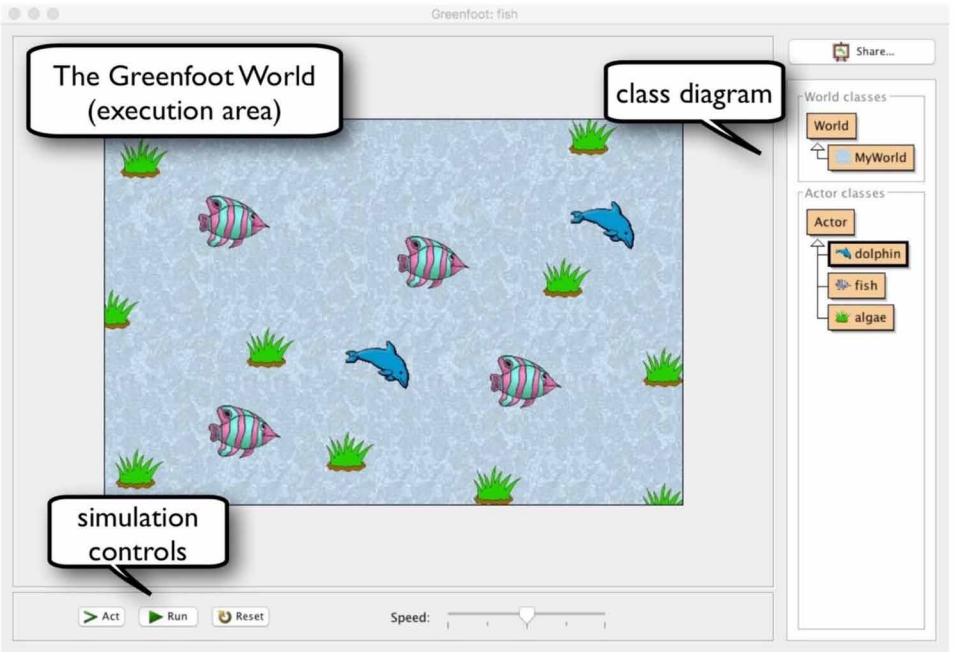


generated from code snippets selected by a learner online, therefore allowing them to learn independently (Harms et al., 2013). They have also looked at how interfaces can support students to reuse code written by others (Gross et al., 2010), as well as considering the effectiveness of learning through code completion puzzles (Harms et al., 2015).

Greenfoot

Greenfoot (http://www.greenfoot.org/) is an educational IDE which started development in 2004 and was published in 2006 (Kölling, 2015). Greenfoot was designed to teach young people to learn to program through the creation of games and simulations (Kölling & Henriksen, 2005). As it is based on Java, there is no upper age limit, however the recommended lower limit is 14 years because of possible difficulties with syntax. Thus, its target audience centres on high school and beginning university students. Code written in the Greenfoot IDE executes in the Greenfoot world, which is a 2D interactive, graphical environment (see Figure 7). The Greenfoot world is inhabited by "actors": in order to get the actors to carry out a behaviour, students program the actor's "act" method. Clicking on the "Act"

Figure 7. The Greenfoot interface



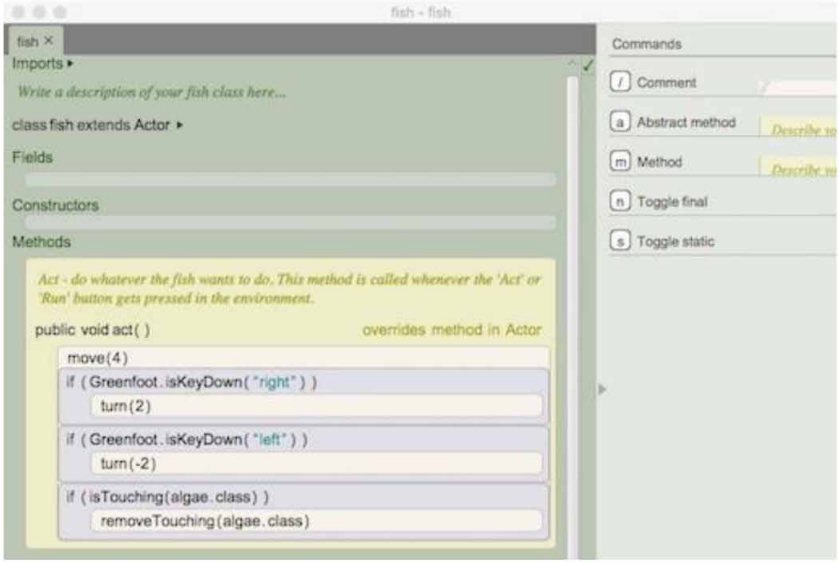
button will call this method once, while clicking on the "Run" button will call the method indefinitely.

In earlier versions of Greenfoot, young people wrote their code in Java. Whilst this is still possible, Greenfoot 3 provided additional possibilities for coding with the introduction of the Stride programming language and, more importantly, a frame-based editor for writing Stride code (see Figure 8).

The frame based editing environment allows novice programmers to write code by selecting from a series of frames using keyboard shortcuts (shown on the right). Frames often contain slots, which can be completed with further frames (e.g. in the case of, say, a nested if statement) or by typing in text, such as object names, etc. Within the frames, the Stride editor allows code to be inserted using code completion, where typing the first letters of, say, a method, will generate a list of methods starting with that prefix. The use of frames allows the system to use contextual cues in generating possible completions.

These particular features of the frame-based editor aim to reduce syntax errors. At the same time, programming in Stride allows students to learn the semantics of Java programming before moving to programming in Java. The Greenfoot environment allows code to be converted from Stride to Java, and vice versa, providing support for the transition from a more tightly constrained language (Stride) to one which has similar semantics, but requires syntactical knowledge (Java).

Figure 8. Frame based editing in Greenfoot



In terms of its motivational hook, Greenfoot is perhaps more domain agnostic than some of the other environments described in this chapter. Although it can support the creation of games, and its creators recognise the pull of games for some students, it is positioned more as a "micro-world meta framework" (Kölling, 2010) which allows for a variety of micro-worlds to be created, spanning a range of domains from games to simulations and visualisations. As such, it can cater for students with diverse interests, and can be applied to topic areas for which games may not be the most appropriate form of interaction.

In terms of pedagogical model, Greenfoot takes a very different approach to the teaching of programming, where it is common to have students write a very small program and observe the results. Instead, Greenfoot students are given a pre-implemented and compiled scenario. They can begin by executing the scenario to see how it works, and then augment the scenario by creating objects and adding them to the world. Doing so allows a number of fundamental OO concepts to be introduced prior to having students actually write code, an approach akin to "game first programming" (Goschnick & Balbo, 2005). At a later stage, students can go on to write their own scenarios in either Java or Stride.

Greenfoot has a number of strengths, which stem primarily from its adaptability. Because it is based on Java, it has both speed and power, runs on hardware typically available in classrooms, is multi-platform, and can benefit from the infrastructure which already exists for Java. Again, because it is Java based, young people are unlikely to "grow out of it" as quickly as some other development environments and when they do, they may be able to transition to other Java based IDEs if they wish to continue to learn to program, having already gained a certain fluency with Java.

Greenfoot has a number of features which provide a sort of "ecosystem" around the development environment itself, and have allowed for the development of an online community in much the same way as Scratch. In addition to standard materials such as tutorials, a website and a textbook (Kölling, 2016), learners can upload scenarios which can be rated by other users, who can also leave comments. Greenfoot also provides support for instructors in the form of the *Greenroom*, where they can engage in discussions and share scenarios, thereby reducing the overhead involved in using Greenfoot, and giving them access to scenarios that they may not have been able to create themselves.

Flip

The Flip environment (http://www.flipproject.org.uk/) takes a very different approach to introducing programming than the other systems, hence a certain amount of background knowledge is necessary in order to fully understand the context in which Flip operates.

Context of Flip

Rather than a standalone IDE, Flip is a programming environment which integrates with an existing game creation toolset, which itself fits within an existing commercial game. Compared to the other environments described in this chapter, this sets it apart, and gives it distinct advantages and disadvantages, described below.

The development of Flip took place within the context of our research on game creation by young people. Our initial game creation research took place using a commercial game, Neverwinter Nights (released in 2002). Flip was designed to integrate with Neverwinter Nights 2 (or NWN2), the successor to the original game, released in 2006. NWN2 is a third person perspective role playing game (RPG) based on Dungeons and Dragons rules[1]. Players explore a large fantasy world and take part in a complex interactive story with multiple branching plots and subplots. Although the game has battle scenes and fights, the narrative component of the game is a prominent and important element, and as the story progresses, the choices that the player makes will have an effect on how the plot unfolds at any given point.

The interesting thing about NWN2 from our perspective was that it shipped with the Electron toolset, a professional quality game development application which was used by the developers, Obsidian Entertainment, to build the game itself. Games enthusiasts used the toolset to create modules for the game, which other players could then download and play, a practice called modding. Because they were using the developers' own tools and resources to design new 3D worlds and adventures, their modules could potentially have a professional quality that is indistinguishable from the content originally contained in the commercial game, and for many, this was a huge motivation.

We used the Electron toolset with hundreds of young people aged 10-16 years old over a number of years and in a number of contexts (school computing lessons, after school clubs, summer holiday workshops, etc.) to help them create their own 3D role playing games. The games created involved significant effort on the part of the young people, with very sophisticated (and often quite humorous) plots, often with hundreds of lines of dialogue for their characters. Because the stories are interactive, character actions and behaviours required some programming. As such, this opened up opportunities for learning both about story creation and programming.

During that period, our research focussed on the learning benefits which could be gained from empowering young people to create their own commercial quality video games. In addition to programming (Howland, Good & Robertson, 2006), we had a particular interest in the ways in which interactive storytelling, of the type found in these role-playing games, could be used to foster young people's narrative development (e.g. Robertson & Good, 2005; Robertson & Good, 2006) and specific literacy skills (Howland, Good & du Boulay, 2008).

However, despite all of the advantages offered by working with a commercial toolset, the primary disadvantage was that it was not designed for young users, nor was it designed for educational purposes. Nonetheless, the Electron toolset, in addition to being available to the public, was a very open-ended system, which included a plugin interface: essentially, the toolset's capabilities can be extended through plugins written in C#. This led us to develop a whole series of plugins designed specifically to help young people with game creation tasks:

- Adventure Author, which can support the creative task of game design (Robertson & Nicholson, 2007);
- Narrative Threads, a suite of tools to support young people's narrative development through game creation (Howland, Good, & du Boulay, 2013);
- Flip, which supports programming tasks, described in this chapter (Howland & Good, 2015; Good & Howland, 2017).

In the next section, I will illustrate the initial stages of creating a game with the Electron toolset, prior to using Flip. I then go on to describe the Flip programming language, and how it integrates with the toolset environment.

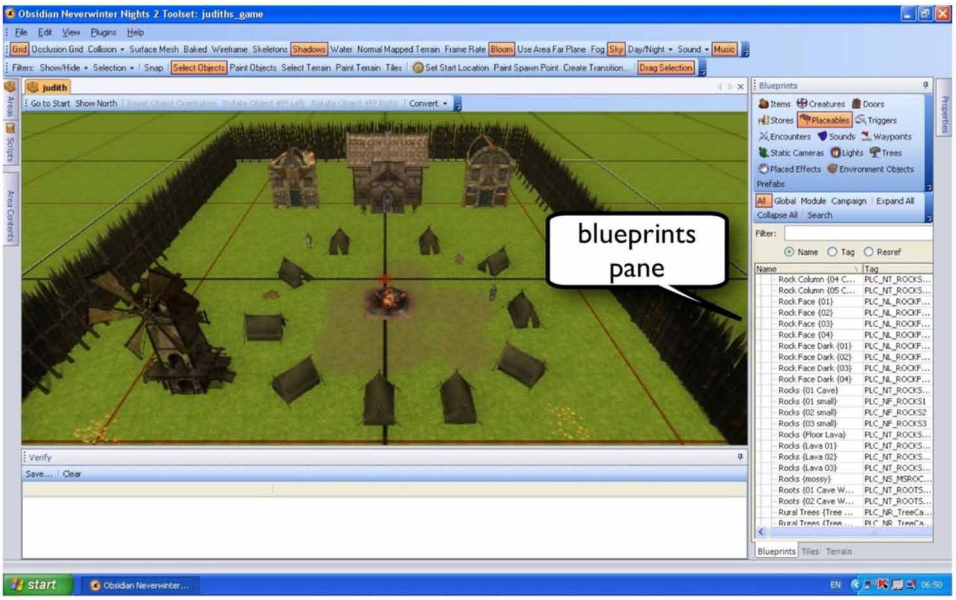
Creating a Game Using the Electron Toolset

The Electron toolset is a graphical environment which provides a grid representation of the 3D game world, and a number of tools for creating a working game. The first step in creating a game involves choosing a setting (referred to as an 'area'). NWN2 allows one to create indoor and outdoor areas and, by customising them, transform them into forests, deserts, castles, dungeons, etc. At a later stage, game designers can add additional areas to their game, and link them together, creating a large world of different types of areas which the player will be able to explore.

Figure 9 shows an area in creation. Various mouse controls allow one to zoom in and out of the area, rotate the area, and pan across it. A new area, at least an exterior one, is a flat, featureless piece of land, so the next stage in game creation is typically to "terraform" or landscape the area: creating mountains and gulleys, and adding landscape features. Once this has been accomplished, discrete objects such as trees, houses, walls, fences etc. can be added by selecting them from the blueprints pane on the bottom right-hand side and dropping them into the area.

The next step is usually to add characters to the area. The toolset provides a wide range of characters, including animals, monsters, mythical creatures and humans. At this stage, the game has the look and feel, if not the interaction, of a commercial game, and young game designers are often keen to view their work in the 3D game

Figure 9. The Electron Toolset with a game being created



world. The game in creation in Figure 9 is shown in Figure 10 as it appears in the game world.

Once characters have been added to the game, conversations can be created for them, and here is where the real creativity of the game creators can begin to shine. Conversations are written as exchanges between the player and any non-player character (NPC). Conversations can be branching, in other words, the game designer can write conversations in which the player, when he or she plays the game, has a choice of responses to give to the NPC. These responses will, in turn, determine how the NPC responds. When the game is being played, the player can approach a character and click on the character to start conversing with it. The example below, written by Zoe, a 10 year old game designer, will help illustrate the concept. Figure 11 shows the conversation as written in the toolset. The conversation is represented as a tree structure, with the indentation representing the conversational turns that can take place between the player and the NPC. Note that turns at the same level of the tree will be represented as choices that the player can choose between in the game world (as shown in Figure 12).

Figure 12 shows part of the conversation as it appears to the player in the game world.

Although the toolset allows for the creation of games of commercial standard and complexity, learners only need around 5 minutes of instruction to begin working

Figure 10. The game as it would appear when being played

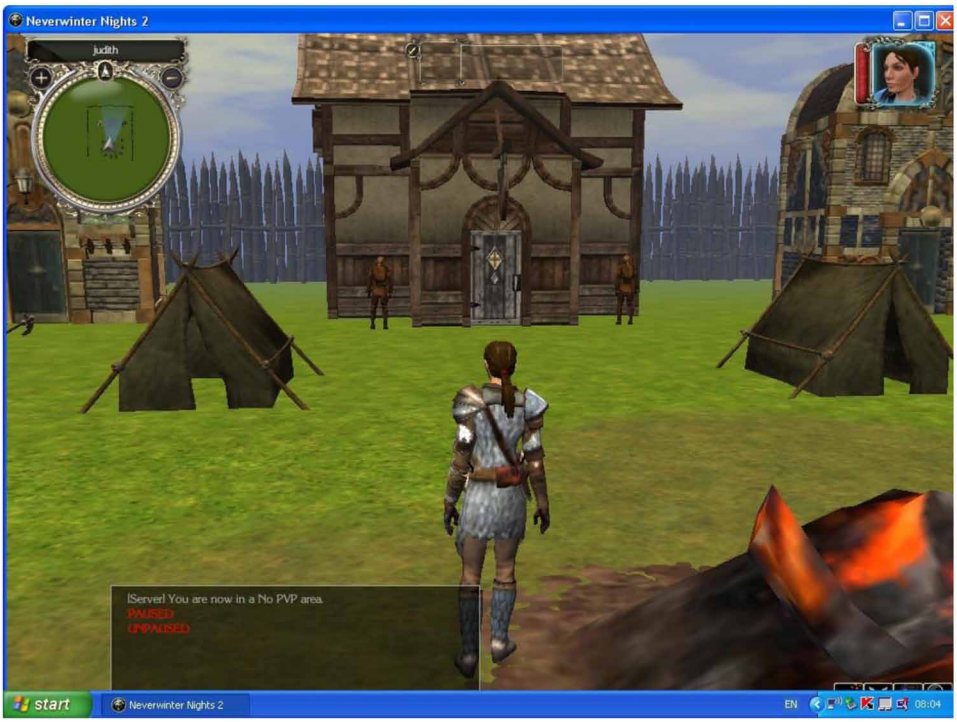


Figure 11. Zoe's conversation in the toolset

NPC: *Hello Abigail Shephard, my name is Legend. Why on earth have you come into my territory? You don't belong here!*

Player: *Okay Legend, you don't understand that you don't have everything to yourself! This is mine too and you're not going to stop me from tresspassing!*

NPC: *Well I am afraid Abigail, that you can't for it is mine!*

Player: *Hmmph! I don't think so Legend!*

NPC: *Oh I do think so Abigail! You are most rude you know, one cannot talk to you without being answered back! And there is a most dreadful tone in you're voice! Calm down girl!*

Player: *Why should I talk to you, know-it-all? You're so posh and polite and it bugs me! I hate goody-two-shoes! Bug off!*

Player: ... Was that my phone ringing? I am sorry I gotta go... Ahh! [END]

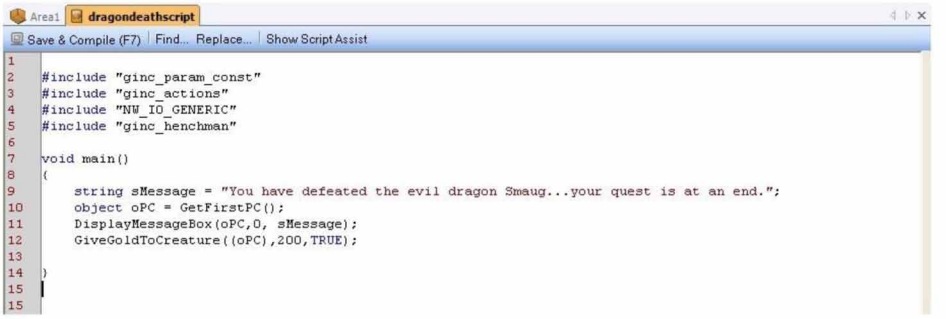
on their own 3D worlds, complete with landscaping, props, characters and other points of interest. However, perfecting one's game can take many hours, days, or weeks, in fact, as much time as one wishes to spend on the task. As such, it is an ideal illustration of the "low floor/high ceiling" effect (Papert, 1980), where access to the toolset is within the reach of all young people who attempt it, thus, it has a low floor, but the fact that it is a professional toolset means that young people are

Figure 12. Zoe's conversation in the game world

unlikely to reach its ceiling, in other words, that they will have goals that they wish to accomplish that the toolset cannot support.

As can be seen above, and in contrast to the programming environments examined earlier in this chapter, it is not necessary to start with programming in order to create a basic game. The design of an area, and any objects and characters in it, can be carried out in a primarily graphical way, by dragging and dropping items, and changing their properties, similar to the drag and drop component of many standard GUI (graphical user interface) tools. Conversations can be written and then attached to characters. Because we are using a commercial games engine, characters already have a number of basic pre-scripted behaviours, for example, walking, running, fighting, attacking, conversing. It is only when sequences of such events need to be set up, when the game designer wants to attach behaviours that are different from those already attached to a particular character, or when she wants to include novel behaviours that programming becomes necessary. Unfortunately, the toolset's inbuilt scripting language, NWScript, is very similar to C in terms of syntax and complexity (see Figure 13 for an example). As such, young people are almost invariably intimidated and frustrated by this element of the game creation process, and are unable to proceed without substantial help from experts.

Figure 13. An NWScript example



On the other hand, the fact that the need for programming occurs comparatively late in the game creation cycle is invaluable for motivation, and gives users the incentive to persevere with the activity. When they ultimately do need to begin programming, their investment in their game world and the narrative they are constructing gives them some drive to try to achieve programming outcomes. After all, a quest to steal the dragon's treasure is not much fun if the dragon hasn't been programmed to defend it!

The Design of Flip

Through the numerous game creation events we have run, we have observed that young people can specify computational rules whilst describing narrative gameplay elements in the context of an informal conversation. However, users often require prompting before they can fully and completely specify the event which might trigger the action, the conditions upon which it is dependent, stopping conditions in the case of a loop, etc., meaning that they face difficulties even before they hit the barrier of NWScript's intimidating syntax. Flip is designed to support young people in moving from their intuitive understanding of narrative events in gameplay towards a computational understanding. In this way, we introduce computational concepts through a motivating activity, and offer a route to computation through the narrative understanding which users have already developed about their game.

A key requirement for Flip was that young people without programming experience should be able to use it to create the events they wanted in their games. However, empowering them to achieve their game creation goals would not, by itself, be sufficient. The environment should also improve users' understanding of the computational processes and concepts which underlie those events.

One of the important features of Flip, and which distinguishes it from other programming languages for novices, is its use of a natural language representation alongside a blocks-based representation. Whilst we employ programming blocks similar in style to those used in Scratch and Alice, we aimed to capitalise on our learners' intuitive understanding of the story events which they were building into their games. We hoped that providing our learners with a dynamically updating natural language description of their computational events might give them a bridge between the narrative description of their story events, and the computational specification of these events, which is necessary for their execution in the game world. Our initial design work therefore focused on the graphical programming blocks, and the functionality of the natural language representation.

Flip was designed using a participatory design methodology, involving learners from the very start. Our approach was based on the CARSS framework (Good & Robertson, 2006), which helps researchers to conduct participatory design by considering the *context* in which the software is to be used, the *activities* which can be carried out during the design phase, the *roles* which the design team members must take on, the necessary *skills* which the team members must possess, and the *stakeholders* involved in the project.

We carried out an extensive range of participatory design activities, focussing on different aspects of the Flip environment, which we describe briefly below. In terms of the graphical programming blocks, we designed a low-fidelity prototype of the Flip language and carried out user testing with young people from a local high school and in a summer holiday workshop context (Good & Howland, 2017). We also asked young people to design their own visual representations for different rule elements (Good & Howland, 2017). When designing the natural language representation element of Flip, we observed non-programmers as they used Inform 7 (Nelson, 2006), a natural language based programming environment for writing interactive fiction (Good & Howland, 2017). We also gathered data on the way in which young people naturally describe game-based events which involve computational concepts, by asking them to observe game sequences and write rules to describe the behaviour that they observed (Good, Howland & Nicholson, 2010). Taken together, we used this data to develop a set of design principles which informed the design of the Flip language, described in (Good & Howland, 2017).

The Flip Language

As described above, the Flip language is a plugin to the existing Electron toolset and replaces the need to use NWScript, the inbuilt scripting language. Flip essentially acts as an overlay to NWScript, whereby scripts written in Flip are translated into NWScript and subsequently interpreted by the game engine and executed in game.

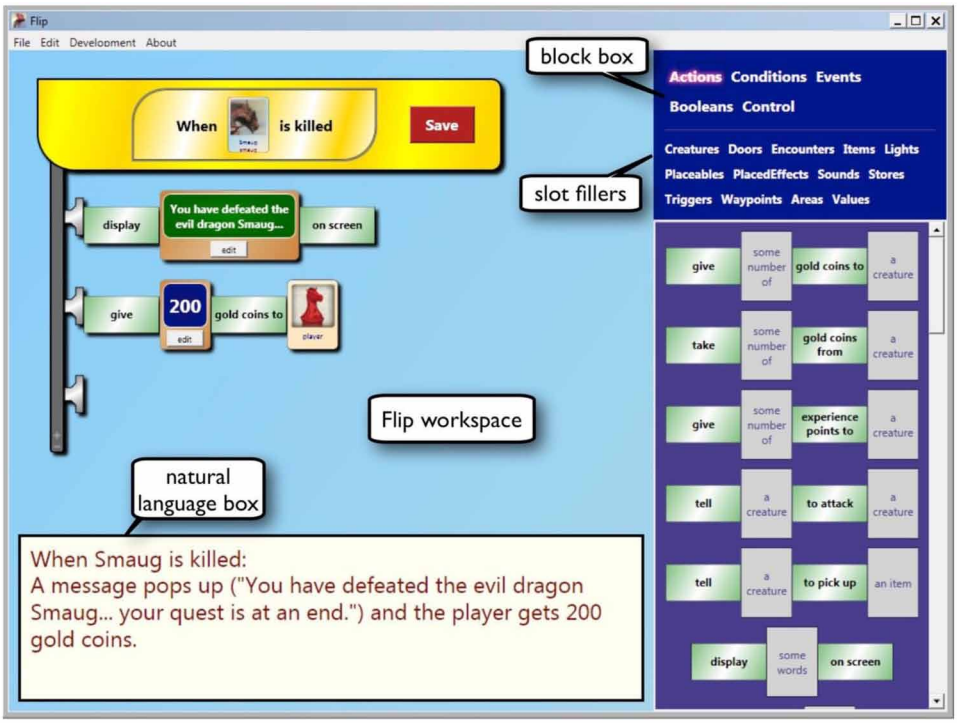
So while young people continue to create areas and customise them with objects and characters using the toolset, they use Flip to add custom behaviours and events to their games.

Flip is an event based language, a commonly used paradigm for developing interactive, game-based applications. All programs must start with a trigger, for example, "When game starts", "When creature is killed". This can then be followed by a single action or a series of actions, which can incorporate typical programming structures such as Boolean logic, conditionals, etc. Figure 14 shows an overview of the Flip interface (note that the script shown is functionally equivalent to the one shown in Figure 13).

On the top right-hand side, the 'block box' contains the blocks used to create scripts, organised by computational category (and colour-coded by type): Actions, Conditions, Events, Booleans and Control. When one of these categories is selected by the user, it is highlighted, and the available blocks in that category are displayed in the bottom right pane (in the example in Figure 14, possible 'Actions' are displayed).

Most blocks have slots, which must be filled by objects of a certain type. These slot fillers are usually objects in the user's game, for example, creatures, items or

Figure 14. The Flip interface



the player. The slots are similarly colour coded to indicate what types of slot fillers are permissible, with empty slots showing, in natural language, the specific type of slot filler required (which effectively constitutes a sort of type system). The choice of slot fillers is shown just below the five computational categories. Although the number of choices may seem elaborate, and the categories confusing, it was necessary to maintain coherence with the NWN2 categories used in the Electron toolset, so young users will be familiar with these category names through the use of the toolset earlier on, when they constructed their areas and populated them with objects.

The Flip workspace is shown on the left-hand side of Figure 14. Scripts in Flip always start with an event block, which contains an event slot to specify the conditions under which the script will execute. The Spine is where the script is composed, by attaching block to the pegs. As blocks are added, the spine can be extended indefinitely to accommodate further blocks.

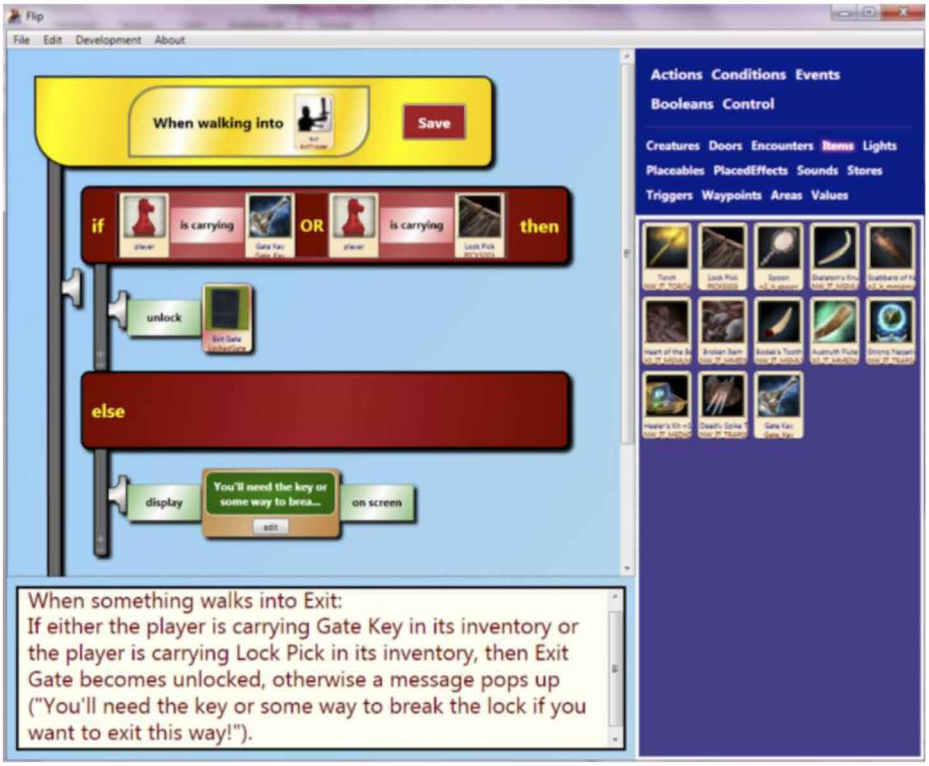
Just under the workspace is the natural language (or 'plain English') box. As the learner builds up a program in the workspace, a natural language description of the program appears here, and is dynamically updated with each change to the program. Given that learners will have, at the outset, imagined their story event in natural language, we designed the natural language box to act as a bridge to the computational world, allowing learners to check, in natural language, that their program will function as intended. Although, our original aim had been to allow learners to edit both the blocks-based and the natural language descriptions, our participatory design work highlighted a number of issues inherent in using natural language for program generation, described in detail in (Good & Howland, 2017).

Figure 15 shows an example of a program with an "if… then…else" statement containing a Boolean conjunction. Control blocks snap onto the spine in the same way as action blocks. In this screenshot, the 'items' category is highlighted in the slot filler pane, meaning that all of the items which have been placed in the user's game also appear here, available for use in Flip scripts.

Only those blocks attached to the spine will execute when the game is loaded and run. This means that learners can experiment with programming blocks in the workspace, dragging out various ones and playing around with them before deciding whether or not to incorporate them into their scripts by snapping them onto the spine. Any elements attached to the spine will be shown in natural language in the 'plain English' box, even if the script is incomplete.

Unlike the other environments described in this chapter, and as mentioned previously, Flip was designed to work in conjunction with a pre-existing game creation environment and game engine. This has placed constraints on the design of the language, both in terms of the types of programming structures that are supported by the existing language, and in terms of the types of scripts that young people are likely to naturally write in the course of creating games of this type.

Figure 15. A Program in Flip



An example of the former concerns the distinction between event triggers and conditionals. At first glance, they may seem quite similar, and indeed in some cases, can act as alternative ways of describing the same event (e.g. "when the dragon is killed", "if the dragon is dead"). If we were unconstrained by the execution environment, we could have made the decision about how to best represent events and conditionals from a purely pedagogical perspective, i.e. in terms of what might be the most comprehensible to novices writing their first programs. In reality, NWN2 and NWScript operate with a number of predefined event hooks (e.g. *OnDeath*, when a creature dies, or *OnClientEnterScript*, when a game area is entered). In order for us to be able to attach Flip scripts to the appropriate script slots so that they would work in game, we needed to create equivalent Flip events for the predefined event hooks in NWScript.

We were also led, to a certain extent, by the types of programming constructs which are likely to occur naturally when young people create games of this type. While we see game construction as an ideal opportunity to engage young people in basic scripting tasks, we were also wary of turning it into an artificial programming

activity, and trying to impose programming tasks in the environment which might somehow break the flow of the narrative goals which young people might be trying to achieve. Loops are one such example. Currently, most of the narrative activity in the game has been focused on the interaction between the player and non-player characters (NPCs), and the need for loops does not arise. Although this means that the programming language is, in some sense, fit for purpose, it does mean that Flip can only cover the introductory elements of a computational curriculum.

We carried out a number of evaluations of Flip, reported in full in (Howland & Good, 2015; Good & Howland, 2017). A two-hour observational study in a secondary school involving 21 pupils allowed us to look in depth at the ways in which users interacted with the Flip interface, and how the features identified as important in our design principles were used by our target users (Good & Howland, 2017). A second, longitudinal study looked at the use of Flip over the course of a game creation project in two contexts: a holiday workshop and a secondary school setting (Good & Howland, 2017). There were differences in results across the two contexts (which was to be expected, given that participation in the holiday workshop was voluntary, while the school study was part of the curriculum for all pupils). However, results were positive overall in terms of ease of use (as evidenced by usage logs as well as interview and survey data), with most learners able to show evidence of an understanding of the role of the various aspects of the Flip interface, as well as the underlying computational concepts. One interesting finding concerned the natural language box, where it was used by individual learners to read the natural description and check whether the script described their intentions accurately. Some learners mentioned that it was particularly useful for more complex scripts. In other cases, the natural language box was used in collaborative settings, with learners paraphrasing the contents in order to explain their script to a teacher or peer or, conversely, with learners first reading another person's natural language description in order to understand the code and help them to debug it.

In a further longitudinal study with 56 secondary school pupils, reported in (Howland & Good, 2015), we again found that the majority of pupils were able to use Flip to create scripts to create interactive events in their games. Furthermore, we examined the impact of using Flip on pupils' computational understanding outside of the game creation environment, and found a significant improvement in their ability to express computational rules and concepts after using Flip (as measured by a pre-/post-test). A further, interesting finding was that girls wrote more, and more complex, scripts than did the boys, most likely because their narratives were more developed and therefore required the scripting of more complex behaviours as a result.

Finally, we were encouraged by teacher comments which suggested that, not only were pupils starting to learn some basic computational skills, but that the

natural language box allowed pupils to better understand the scripts they created, and supported them as they worked collaboratively on tasks such as debugging:

There were a lot of kids talking about it [the output of the plain English box], but the times that stick in my mind were when things weren't working. It was quite often when kids were trying to solve problems or they were helping each other out. They looked and they read what that was saying or would speak, "Oh look, look what you're doing". And quite often, the kid helping the other kid would actually look at the English box first, as if they were trying to figure out what they were wanting to do and then looking up to see what they'd done.

Although the teacher acknowledged that blocks-based programming languages are designed to be user friendly, he nonetheless felt that a natural language equivalent may boost learners' confidence in their scripting ability:

There still may be the underlying lack of confidence in creating the programming structure, even though it's blocks, and using the English as that confirmation to say, "Yes, that is what I want to see."

The teacher also felt that the natural language box might have further, more generalizable benefits in helping pupils gain skills in using computational terminology:

It was really useful to be able to do these more technical aspects of creating a game without actually them realising they're programming and scripting, but also them developing the language and confidence to say, 'Oh, if I do this, then I'll have to do that' and them actually using the terminology without thinking about it.

I think generally just the comfort with using some technical terms and language, because I was able to direct people to set menus and things without really having to describe it. It was actually a greater confidence in using the language.

NOVICE PROGRAMMING ENVIRONMENTS COMPARED

The programming environments above span a range of genres and, in some sense, provide good complements to each other. In terms of age, all target a range of late childhood to early adulthood, with Scratch targeting the youngest users (and, with the advent of ScratchJr, even younger users), and Greenfoot the oldest. Syntax appears to be one of the factors which determines the appropriateness of language to age range: the closer the language is to a fully-fledged programming language,

the less appropriate it will be for younger users. ScratchJr removes the concept of variables from Scratch in order to service even younger users.

The environments also differ in terms of their relationship to existing programming languages. Although built on NWScript, which is similar to C, Flip is not designed to be a first step to a general-purpose programming language, nor is Scratch (although in many current computing curricula, it is often used as the first language that pupils encounter, before moving to more traditional languages). Alice and Looking Glass model OO concepts, and Alice at least is designed as a precursor to Java, with the latest version of Alice having the ability to show the generated Java code in a window alongside the Alice code. Furthermore, an Alice plugin for the NetBeans IDE allows an Alice 3 project to be transferred to NetBeans where it can be enhanced by editing the converted Java code. Programming in Greenfoot can be carried out in both Java and in Stride (the Java-like language included with Greenfoot's frame based editor). As such, Greenfoot provides both the most direct link to a general-purpose programming language, and the most support for making that transition.

Some of the designers of these systems, such as the authors of Greenfoot, have consciously set out with a list of design goals which have helped focus the design and implementation of their system, for example, "support engaging functionality", in other words, allow learners to implement features in their programs which they find compelling, such as sound, graphics and animation, or allow social interaction and sharing (Kölling, 2010). Other researchers (Good & Robertson, 2006) have remarked on the sorts of motivational and learning affordances which existing systems seem to offer young users, often unintentionally, a point also highlighted by Resnick (2017). Still others (Petre & Blackwell, 2007) have examined the patterns of activity engaged in by young users, noting those which, although occurring in informal contexts, are nonetheless characteristic of software development. In some cases, researchers have looked at how they can enhance the success of existing systems, such as Alice, or Neverwinter Nights 2, to offer further improvements, either in terms of motivation for specific user groups, in the case of Looking Glass, or further pedagogical support, in the case of Flip. What is reassuring is that for all of the environments described above, the effects which have been designed for appear to be playing out in practice. What is also interesting is that although the environments described above may differ, the benefits they offer are remarkably similar.

Finally, there is the question of domain, or the purpose for which the languages can be used. None of the programming environments are designed to be truly general purpose, rather their aim is to provide a "motivational hook" which can encourage learners to engage with programming more as a means to an end, rather than an end in itself.

REFLECTIONS ON ENVIRONMENTS

Below I reflect on a few of the themes which cut across the environments described in this chapter before going on to consider possible implications for teaching, research and further development in the area. It is interesting to note that the three themes identified in 2011 remain just as relevant today.

Learner-Centred and Learner-Led

The designers of all of these programming environments share a common vision that the environments will, or at the very least can, be used in a way that is learner-centred and learner-led, rather than curriculum-driven. As such, they are likely to be more motivating than traditional methods of teaching programming. As du Boulay (personal communication, 12 November 2014) notes, "The problem with programming is that you have to convince people that they want to solve problems that they haven't yet imagined, and then you give them a tool to solve these problems that they didn't actually have before." In contrast, in the environments described above, learners can start with a topic of personal interest, and use this interest to motivate their learning, bringing in computing topics on an "as needed" basis, rather than in a strict order. Resnick et al.'s (2009) example of the learning of variables in order to keep score in a game is a good example of this type of learning. To take an example from Looking Glass, a young person may be switched on to programming by a peer, and decide to give it a go because she is motivated by the idea of making her own story-based movie. She chooses the focus for her story, and is able to search for code snippets that will accomplish the behaviours that she requires for her project. Better still, rather than generic tutorials describing the functionality of Looking Glass, any tutorials she requires will be specifically tailored to the code snippets she is working on at that particular point in time. The contextual nature of this use of support materials is compatible with the way modern programmers work with online documentation, where languages such as Java and C# have thousands of classes and interfaces, and many more methods.

Rapid Pay Off

In all of the environments, seeing a very quick return on investment seems key in establishing and maintaining motivation. In 1993, in a debate entitled, "Should we teach students to program?", Soloway and Guzdial described difficulties in learning to program as follows:

Learning to express oneself in a GPPL [General Purpose Programming Language], as they are currently conceived, requires a steep learning curve: writing a 100-line program is much harder than writing a 10-line program; writing a 1,000-line program is much, much harder than writing a 100-line program. And, let's be honest, to make the computer truly sing and dance, one needs to write significantly sized programs. What practical program can someone who finishes -- even successfully -- Computer Science 101 actually write? (Soloway, 1993, p. 22)

The picture had already changed significantly in 2011, and it continues to change. Young people can now easily write a small program to make a computer sing and dance, quite literally, in a few minutes. The "computational floor" has been lowering dramatically, allowing easier access to programming for learners. And yet, the ceiling has stayed high, meaning that the challenges available to learners as they grow in competency have remained. Armed with initial feelings of success, young people are more likely to continue into the complexities of programming with greater confidence, and persevere when things become difficult.

Rapid Feedback Loop

As has been mentioned, all of the environments described in this chapter comprise a programming language which is tightly integrated with an execution environment, either 2D or 3D. Because of this, it's very easy to (literally) see whether a program is behaving as intended (or not), and make any necessary changes. Although this is only anecdotal, I would suggest that the ludic nature of the execution environments probably encourages a wider range of hypothesis testing than would normally be the case with traditional programming languages (e.g. "Ok, now the dragon attacks the giant whenever he gets near the treasure chest. But what would happen if a rogue bear arrived on the scene and stole the chest before the giant arrived?"). These types of scenario are easy (and fun) to try out, and many systems, for example, Scratch, have been set up explicitly to encourage this sort of tinkering (Maloney et al., 2010).

Creation of a Valued Artefact

Constructionism is built on the premise that learning takes place in, and is almost a by-product of, contexts in which one is engaged in the building of a public entity or artefact (Papert & Harel, 1991). It stands to reason that motivation can only be heightened when the artefact being built has a particular meaning for the individual and their immediate peer group. Kelleher has engaged with this issue from a gender perspective, arguing that storytelling has a particular resonance with girls, and is likely to inspire girls to take an interest in programming. Kölling has taken a different

tack by trying to maximise personal choice, recognising that not every young person will be motivated by creating a game, and should have a broader arena of possible choices. In all of the novice programming environments described however, the assumption is that young people will be able to start from the perspective of their own interests, and have the ability to create and/or make use of their own media so as to produce something that they and their peers will find to be of value. In this respect, novice programming has come a long way from the days of "Hello world" and the Fibonacci sequence.

Recognition of the Importance of Peers

In all of the environments described, peers play an important role. Not only can the development of programs itself be a highly social process, there is something intrinsically motivating about building something that you can share with your peers once it is completed. In addition to local communities of friends and fellow learners, the internet has made it easy to form virtual communities which are every bit as powerful in their support. As learners develop and share their programs, they are also gaining an appreciation of the intrinsic value of a community of users, and recognising the contribution of individual users' skills. This in turn can foster an appreciation for teamwork and collaboration in multiple guises. I don't think we have begun to scratch the surface of examining the types of collaboration which exist between learners, and the types of learning benefits which can accrue as a result, and I therefore consider how this might be more fully supported in the section on "Implications for Teaching, Research and Further Development".

Changing Times

In 2011, I noted that novice programming environments seemed to have "come of age", in that they "are freely available to young people across the globe, and have broad fan bases, who enthusiastically share not only their latest creations, but also their knowledge and understanding with others." Since that time, the use of novice programming environments has only increased, in some cases, seemingly exponentially. As an example, community statistics for Scratch (https://scratch.mit.edu/statistics/) show that in the month of January 2011, the number of new projects created was 56,575, while in the month of September 2017, 703,883 new projects were created. It is an open question whether the introduction of programming in school curricula worldwide is the main significant contributing factor (e.g. the English national curriculum introduced statutory computing programmes of study for all educational key stages, i.e. from age 5 to 16 (Department for Education, 2013).

Indeed, since 2011, computing for novices, particularly children, has increasingly taken centre-stage. Computing features much more prominently in the school curricula of many countries (see the special issue in ACM's Transactions on Computing Education, entitled, "Computing Education in K-12 Schools from a Cross-National Perspective", 14(2), for some examples), and there has been a proliferation of new programming environments for novices. Many of these environments stem from Blockly (https://developers.google.com/blockly/) which itself borrowed heavily from Scratch's codebase. Such environments include MIT's App Inventor (http://appinventor.mit.edu/explore/), and Code.org (https://code.org/).

Perhaps one of the biggest developments in recent times is the rise of hybrid environments, with environments providing increasing support for viewing and/or manipulating code in more than one format. As we have seen in this chapter, advances in Greenfoot allow users to program in both Stride and Java, potentially supporting a progression from learning Java semantics in Stride before encountering Java syntax. Alice 3 similarly aims to support progression by allowing learners to view the Java version of their Alice code, and then transfer their Alice code into Java. On a slightly different tack, Flip provides a natural language view of the code which, whilst not manipulable, aims to provide an additional representation which can support computational understanding. This focus on supporting progression between different languages (in particular, moving between blocks based and text based languages) is also evidenced in environments such as PencilCode (Bau et al., 2015), Tiled Grace (Homer & Noble, 2014) or DrawBridge (Stead, 2016).

Implications for Teaching, Research and Development

In 2011, I suggested that, from a teaching perspective, "the lessons to be learned from these environments would suggest fitting the curriculum to the experience of building a reasonably large sized project, rather than the other way round". I will revisit the two specific issues I identified under this heading: that of focusing on projects, rather than topics, and considering the processes involved in project based work, rather than just the outcomes. In so doing, I will consider the extent to which these topics are addressed in current methods of teaching programming.

1. Projects, not Topics

It is easy to fall into the trap, when planning a new course in computing, of focusing on a list of topics that need to be covered in a linear order. In many cases, it would seem that a behaviourist mode of curriculum-centred instruction is still de rigueur in many institutions, and it is not surprising that many students fail to find computing motivating or relevant. The environments described in this chapter all facilitate an

approach whereby a curriculum could be designed around student-led projects, with curriculum topics introduced when required, and evidence of learning gathered from the projects themselves, rather than from a decontextualised, standardised test at the end. Certainly the research described by Resnick et al. (2009) suggests that core computing topics do get covered in the course of project work. Of course, based on their observations, and those of Petre and Blackwell (2007), this approach begs the question, "If students don't know they're programming, does it count?" I would argue that the instructor's role would then become, rather than directly instructing students on topics, one of drawing out the implicit knowledge that the learners have acquired, helping them to reflect on it, and to see how it might apply in other situations. At the same time, there are real concerns as to whether such an approach is sustainable within the strict confines of school curricula, which are increasingly focussed on testing and results. As Resnick (2017) notes, even in the early years curriculum, there is an increasing focus on instruction rather than exploration, with today's kindergartens sometimes referred to as "literacy boot camps". Interestingly, Fincher (2015) considers the parallels between approaches to traditional literacy and the current call for "computer literacy". She describes the many examples of failed initiatives to increase literacy, and suggests that, in considering computer literacy, we should learn from these examples, starting by first articulating what it means to be "literate" and ensuring that our approaches are based on considerable research.

2. Processes and Products

The project based approach, advocated above, leads to a tangible outcome, which has important benefits for students, not least in terms of motivation. However, in addition to outcome, it is equally important to look at the *processes* involved in arriving at that outcome, as they may be equally worthwhile in learning terms. Again, a behaviourist view of learning would suggest that our hopes for our students can be neatly encapsulated in a set of learning outcomes (themselves ideally expressed in ways that can be measured and assessed). However, many of us would agree that the activities involved in computing are broad and far reaching. They involve design and planning, testing and rethinking. We have evidence to suggest that young people willingly engage in these activities in the environments described in this chapter, and it would be worth putting more thought into how we can capture evidence of these activities in an ecologically valid way in pedagogical contexts rather than, again, looking to measure understanding through testing. As a way of stimulating thinking on the sorts of things we might want to look for, I include Petre and Blackwell's intriguing observations on children engaged in informal programming activities, who state that they are:

… able to discuss their goals, actions and artefacts. They can identify and adapt components for re-use; recognize and generalize from patterns in different examples and explain what things do, what they are for, and why they are designed that way (although the explanations may not be in conventional language). …they are able to reason about modifications and consequences, and about interactions between components. They are able to diagnose unexpected behaviours by systematic reasoning and experimentation. Occasionally, they encounter the need to restructure programs. They spontaneously introduce disciplines such as version control, naming conventions, design for re-use and systematic debugging. (Petre and Blackwell, 2007, p. 241)

There is certainly a focus on methods for assessing computational thinking skills (see, e.g. (Grover, Cooper & Pea, 2014)) but, again, Resnick (2017) warns against overly simple quantitative measures of achievement and, instead, advocates a more holistic approach to achievement in line with project-based activities, where we "focus on what's most important for children to learn, not what's easiest for us to measure" (Resnick, 2017, p. 148).

Implications for Research and Development

So what are the implications of these novice programming environments from a research and development perspective? In 2011, I identified three themes. Firstly, I suggested that our research focus should be on investigating the particular characteristics and combinations of features and modalities inherent in modern IDEs, rather than just comparing programming languages (e.g. "visual vs. textual"). Secondly, I noted that modern novice programming environments offer ideal playgrounds for studying collaboration "in the wild". Finally, I suggested that, in conjunction with a focus on collaboration, we should nonetheless continue to study environment development from a cognitive perspective, with a view to best supporting the tasks that individual learners hope to accomplish.

In this chapter, I take a slightly different perspective on these same themes. Firstly, I consider the renewed debate around "blocks based vs. text-based" and whether we might more usefully be focussing on the "space between" and on how to support transition between languages, but also on the particular properties of each IDE in more detail. Secondly, I look at different forms of collaboration in more detail, and consider how they might best be supported. Finally, given the increase in complexity of novice programming environments, at least from a representational perspective, I renew the call for considering the cognitive aspects of accomplishing tasks in such environments, and in ensuring that the respective representations are working together to best support learners.

1. "Visual vs. Textual" Is No Longer a Useful Distinction for Programming Languages, but Neither Is "Blocks-Based vs. Textual"

Having carried out research on visual programming languages back in the day, this feels like rather a bold claim to make. In the 1980s and 1990s, there was a much clearer demarcation between textual programming languages and visual programming languages (even if a fair bit of the debate on comp.lang.visual focused on what exactly qualified as a visual programming language). In the 1990s, a number of empirical studies of visual programming languages were carried out in order to ascertain whether they made the task of programming, and program comprehension easier, particularly for novices (see, e.g. (Blackwell et al., 2001) for a summary of this debate). Today, the focus is much less on programming languages per se, and more on integrated development environments, and these are much more likely to be hybrids, with a mix of text and graphics co-existing quite happily. However, much of the debate around programming languages continues to be focussed on the high-level "blocks vs. text" distinction. Although this is a useful way of categorising languages on a first pass, I maintain that our research needs to be carried out at a finer level of granularity. I made the point in (Good & Howland, 2017) that blocks-based (or visual) languages offer an inherently high level of constraint as compared to textual languages to the extent that programming using graphical languages involves choosing from existing statements, whereas programming using textual languages often involves creating these statements from scratch. Therefore, it may be that any ease of use attributed to graphical languages results from their constrained nature, rather than their graphicacy per se. Greenfoot's frame-based editor offers some of these same constraints and, as such, would provide an excellent testbed for untangling some of the claims around blocks-based vs. text-based languages.

2. Current Environments Are Ideal Playgrounds for Studying Different Types of Collaboration

In each of the environments described above, there has been a focus on the beneficial nature of collaboration for learning, an issue which Resnick (2017) considers in some depth in relation to Scratch. Part of the power of these online environments derives from their ability to offer extensive opportunities for collaboration and sharing in a way which is different from the "enforced" collaboration sometimes associated with more traditional educational configurations. Resnick et al. (2009) have already pinpointed a number of different collaborative relationships which children enter into, and partnerships which they are able to negotiate, including forming online "companies". Wenger's model of communities of practice (Wenger, 1999) could

very usefully be applied to the sorts of activities which are occurring on a daily basis on sites such as the Scratch website.

Petre and Blackwell also note the importance of social networks not just as ways of engaging in collaborative or cooperative programming, but even as "a mechanism for adjusting understanding and correcting conceptual and operational models" (Petre and Blackwell, 2007, p. 242). If one wished to take a more formal pedagogical approach, it would be possible to look at such practices through the lens of the Zone of Proximal Development (Vygotsky, 1978), looking for instances where children are being aided by their peers in mastering concepts which would have otherwise been just slightly out of their reach. Ironically, the educational field is rife with examples of situations where collaboration has been "engineered in", often unsuccessfully: the environments described in this chapter offer rich opportunities for looking at how it arises organically, and how it is sustained.

At the same time as these environments have opened up tremendous *opportunities* for collaboration, it's also important to look at how we might provide non-intrusive *support* for collaboration, both remote and co-located collaboration. One of the unanticipated benefits of Flip's natural language box was that it offered this type of support: by giving young people a language to describe computational concepts as well as their specific programs, it opened up opportunities for spontaneous collaboration and peer support. It would be good, in future, to look at how we can build on such initiatives.

3. However, the Collaborative Shouldn't Crowd Out the Cognitive

Although I have suggested we might fruitfully examine the patterns of activity which occur between learners, this is not to suggest that we do so at the expense of individual cognition and, in particular, at looking at how these novice programming environments function as external representations in supporting cognitive activity. In 2011, I wrote that "there is a need to continue to conduct studies which examine the ways in which learners use such environments, including the ways in which they interact with multiple external representations, move between their code and the execution environment, and generally make sense of multiple data streams". This is all the more important now, given that an increasing number of programming environments offer multiple representations of programming code (e.g. Alice/Java in Alice, Stride/Java in Greenfoot). Not only must leaners be able to understand the mappings between these multiple representations, but they must also be supported to develop what Stead & Blackwell (2014) term "notational expertise".

CONCLUSION

This article has considered the state of the art in novice programming environments, looking at five such exemplars: Scratch, Alice, Looking Glass, Greenfoot and Flip. Although different in many ways, all of the designers share a common aim in involving learners in building and manipulating computational artefacts, and in a way that motivates and empowers them. A number of changes in recent years, both technological and cultural, have led to these environments being truly "people oriented", with young people engaging in programming activities and with each other, in ways that (even) the tool designers may not have been able to fully envisage. While they offer stimulating virtual playgrounds for students to try out creative ideas, I have argued that they also offer rich new tools for educators and researchers alike, and the hope is that innovative pedagogy and research will be built upon these environments.

ACKNOWLEDGMENT

I would like to thank, first and foremost, Kate Howland and Keiron Nicholson, my research fellows on the Flip project and all of the young people and teachers who willingly helped us design and test Flip. I am also thankful to Steve Goschnick and to colleagues at the University of Sussex who helped me hone some of the reflections presented in this chapter. The Flip project was funded by the UK Engineering and Physical Sciences Research Council (EPSRC), EP/G006989/1, I am very grateful for their support.

REFERENCES

Adams, J. (2014). *Alice 3 in Action: Computing Through Animation*. Cengage Learning.

Bau, D., Bau, D. A., Dawson, M., & Pickens, C. (2015). Pencil Code: Block code for a text world. In *Proceedings of the 14th International Conference on Interaction Design and Children* (pp. 445-448). ACM. 10.1145/2771839.2771875

Blackwell, A. F., Whitley, K. N., Good, J., & Petre, M. (2001). Cognitive factors in programming with diagrams. *Artificial Intelligence Review*, *15*(1), 95–114.

Cooper, S., Dann, W., & Pausch, R. (2003). Teaching objects-first in introductory computer science. *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education* (pp. 191-195). New York: ACM Press. 10.1145/611892.611966

Department for Education. (2013). *The National Curriculum in England: Computing programmes of study.* Available from: https://www.gov.uk/government/publications/national-curriculum-in-england-computing-programmes-of-study

Fincher, S. (2015). What are we doing when we teach computing in schools? *Communications of the ACM*, *58*(5), 24–26. doi:10.1145/2742693

Fletcher, G., & Lu, J. (2009). Human computing skills: Rethinking the K-12 experience. *Communications of the ACM*, *52*(2), 23–25. doi:10.1145/1461928.1461938

Good, J., & Howland, K. (2017). Programming language, natural language? Supporting the diverse computational activities of novice programmers. *Journal of Visual Languages and Computing*, *39*, 78–92. doi:10.1016/j.jvlc.2016.10.008

Good, J., Howland, K., & Nicholson, K. (2010). Young people's descriptions of computational rules in role-playing games: An empirical study. In *2010 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 67-74). IEEE. 10.1109/VLHCC.2010.18

Good, J., & Robertson, J. (2006). CARSS: A framework for learner-centred design with children. *International Journal of Artificial Intelligence in Education*, *16*(4), 381–413.

Goschnick, S. (2015). App Review: ScratchJr (Scratch Junior). *International Journal of People-Oriented Programming*, *4*(1), 50–55. doi:10.4018/IJPOP.2015010104

Goschnick, S., & Balbo, S. (2005). Game-first programming for information systems students. In *Proceedings of the Second Australasian Conference on Interactive Entertainment* (pp. 71-74). Creativity & Cognition Studios Press.

Gross, P. A., Herstand, M. S., Hodges, J. W., & Kelleher, C. L. (2010). A code reuse interface for non-programmer middle school students. *Proceedings of the 14th International Conference on Intelligent User Interfaces* (pp. 219-228). New York: ACM Press. 10.1145/1719970.1720001

Grover, S., Cooper, S., & Pea, R. (2014). Assessing computational learning in K-12. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education* (pp. 57-62). ACM.

Guzdial, M. (2004). Programming environments for novices. In S. Fincher & M. Petre (Eds.), *Computer Science Education Research* (pp. 127–154). London: Taylor & Francis.

Guzdial, M. (2008). Paving the way for computational thinking. *Communications of the ACM*, *51*(8), 25–27. doi:10.1145/1378704.1378713

Harms, K. J., Cosgrove, D., Gray, S., & Kelleher, C. (2013). Automatically generating tutorials to enable middle school children to learn programming independently. In *Proceedings of the 12th International Conference on Interaction Design and Children* (pp. 11-19). ACM.

Harms, K. J., Rowlett, N., & Kelleher, C. (2015). Enabling independent learning of programming concepts through programming completion puzzles. In *Proceedings of the 2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 271-279). IEEE. doi:10.1145/2485760.2485764

Homer, M., & Noble, J. (2014). Combining tiled and textual views of code. In *2014 Second IEEE Working Conference on Software Visualisation (VISSOFT)* (pp. 1-10). IEEE. 10.1109/VISSOFT.2014.11

Howland, K., & Good, J. (2015). Learning to communicate computationally with Flip: A bi-modal programming language for game creation. *Computers & Education*, *80*, 224–240. doi:10.1016/j.compedu.2014.08.014

Howland, K., Good, J., & du Boulay, B. (2008). A game creation tool which supports the development of writing skills: Interface design considerations. In *Proceedings of Narrative and Interactive Learning Environments (NILE 08)*, 23-29.

Howland, K., Good, J., & du Boulay, B. (2013). Narrative Threads: A tool to support young people in creating their own narrative-based computer games. In *Transactions on Edutainment X* (pp. 122–145). Berlin: Springer. doi:10.1007/978-3-642-37919-2_7

Howland, K., Good, J., & Nicholson, K. (2009). Language-based support for computational thinking. In *Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, (pp. 147-150). IEEE. 10.1109/VLHCC.2009.5295278

Howland, K., Good, J., & Robertson, J. (2006). Script Cards: A visual programming language for games authoring by young people. In *Proceedings of the 2006 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 181-184). IEEE.

Kelleher, C., Cosgrove, D., Culyba, D., Forlines, C., Pratt, J., & Pausch, R. (2002). Alice 2: Programming without syntax errors. In *Proceedings of the 2002 Conference on User Interface Software and Technology*. Available from: https://uist.acm.org/archive/adjunct/2002/pdf/demos/p35-kelleher.pdf

Kelleher, C., & Pausch, R. (2005). Lowering the barriers to programming. *ACM Computing Surveys*, *37*(2), 83–137. doi:10.1145/1089733.1089734

Kelleher, C., & Pausch, R. (2007). Using storytelling to motivate programming. *Communications of the ACM*, *50*(7), 58–64. doi:10.1145/1272516.1272540

Kelleher, C., Pausch, R., & Kiesler, S. (2007). Storytelling Alice motivates middle school girls to learn computer programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 1455 - 1464). New York: ACM Press. 10.1145/1240624.1240844

Kölling, M. (2010). The Greenfoot programming environment. ACM Transactions of Computing Education, 10(4), Article 14.

Kölling, M. (2015). Lessons from the Design of three educational programming environments: Blue, BlueJ and Greenfoot. *International Journal of People-Oriented Programming*, *4*(1), 5–32. doi:10.4018/IJPOP.2015010102

Kölling, M. (2016). *Introduction to Programming with Greenfoot: Object-Oriented Programming in Java with Games and Simulations* (2nd ed.). Pearson.

Kölling, M., & Henriksen, P. (2005). Game programming in introductory courses with direct state manipulation. In *Proceedings of the 10th ACM–SIGCSE Annual Conference on Innovation and Technology in Computer Science Education* (pp. 59-63). New York: ACM Press. 10.1145/1067445.1067465

Maloney, J., Peppler, K., Kafai, Y. B., Resnick, M., & Rusk, N. (2008). Programming by choice: Urban youth learning programming with Scratch. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education* (pp. 367-371). New York: ACM Press. 10.1145/1352135.1352260

Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The Scratch programming language and environment. *ACM Transactions on Computing Education, 10*(4), 16:2-16:15.

Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. (2013). Learning computer science concepts with scratch. *Computer Science Education*, *23*(3), 239–264. doi: 10.1080/08993408.2013.832022

Nelson, G. (2006). *Natural Language, Semantics Analysis and Interactive Fiction.* Available from: http://inform7.com/learn/documents/WhitePaper.pdf

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. New York: Basic Books.

Papert, S., & Harel, I. (1991). Situating constructionism. In S. Papert & I. Harel (Eds.), *Constructionism*. Norwood, NJ: Ablex Publishing Corporation. Available from: http://www.papert.org/articles/SituatingConstructionism.html

Petre, M., & Blackwell, A. F. (2007). Children as unwitting end-user programmers. In *Proceedings of the 2007 IEEE Symposium on Visual Languages and Human-Centric Computing* (pp. 239 – 242). IEEE.

Resnick, M. (2014). Give P's a chance: Projects, peers, passion, play. In *Constructionism and Creativity: Proceedings of the Third International Constructionism Conference*. Austrian Computer Society.

Resnick, M. (2017). *Lifelong Kindergarten: Cultivating Creativity Through Projects, Passion, Peers, and Play*. MIT Press.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., ... Kafai, Y. (2009). Scratch: Programming for all. *Communications of the ACM*, *52*(11), 60–67. doi:10.1145/1592761.1592779

Rizvi, M., Humphries, T., Major, D., Jones, M., & Lauzun, H. (2011). A CS0 course using scratch. *Journal of Computing Sciences in Colleges*, *26*(3), 19–27.

Robertson, J., & Good, J. (2005). Story creation in virtual game worlds. *Communications of the ACM*, *48*(1), 61–65. doi:10.1145/1039539.1039571

Robertson, J., & Good, J. (2006). Supporting the development of interactive storytelling skills in teenagers. In Z. Pan, R. Aylett, H. Diener, X. Jin, S. Göbel, & L. Li (Eds.), Technologies for E-Learning and Digital Entertainment. Edutainment 2006 (pp. 348–357). Lecture Notes in Computer Science Springer; doi:10.1007/11736639_46.

Robertson, J., & Nicholson, K. (2007). Adventure Author: A learning environment to support creative design. In *Proceedings of the 6th International Conference on Interaction Design and Children* (pp. 37 - 44). New York: ACM Press. 10.1145/1297277.1297285

Soloway, E. (1993). Should we teach students to program? *Communications of the ACM*, *36*(10), 21–24. doi:10.1145/163430.164061

Stead, A., & Blackwell, A. F. (2014). Learning syntax as notational expertise when using DrawBridge. In *Proceedings of the Psychology of Programming Interest Group Annual Conference (PPIG 2014)* (pp. 41-52). Academic Press.

Stead, A. G. (2016). *Using multiple representations to develop notational expertise in programming*. University of Cambridge, Computer Laboratory, Technical Report, (UCAM-CL-TR-890).

Vygotsky, L. S. (1978). *Mind and society: The development of higher psychological processes*. Cambridge, MA: Harvard University Press.

Wenger, E. (1999). *Communities of practice. Learning, meaning and identity*. Cambridge, UK: Cambridge University Press.

Wing, J. (2006). Viewpoint-Computational Thinking. *Communications of the ACM*, *49*(3), 33–35. doi:10.1145/1118178.1118215

ENDNOTE

[1] Note that a successor to Neverwinter Nights 2 was not released.